Towards aspect-oriented programming support for cluster computing

Matthew David Wolenetz Hasnain A. Mandviwala Sameer Adhikari Yavor Angelov
Umakishore Ramachandran Kenneth Mackenzie James Matthew Rehg

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Phone: (404) 894-5136
FAX: (404) 385-2295
e-mail: rama@cc.gatech.edu
WWW URL: http://www.cc.gatech.edu/~rama

Abstract

Interactive multimedia applications (such as audio/video processing) are good candidates for cluster computing. Such applications are best represented as coarse-grain dataflow graphs and are rich in pipelined, task, and data parallelism. Specification of the strategies for mapping computational abstractions to compute nodes, and their plumbing are two important issues in the design of such complex parallel and distributed applications. Due to the varieties of parallelism that are available in such applications, the space of strategies to be explored can be vast. We have developed an aspect-oriented programming language for cluster computing called *STAGES*. This language allows the algorithm design to be disentangled from the connection management and performance concerns. STAGES provides a simple syntax for specifying the connections among threads and data abstractions, and their mapping onto the nodes of the cluster. The current implementation targets the *Stampede* cluster programming library. However, the language is general and can be retargeted to a different set of abstractions.

In this paper, we present STAGES, its implementation, and its utility for mapping complex applications onto a cluster. We also present performance results from exploring the parallelism space for two such applications on a 17-node cluster of 8-way SMPs (Intel Xeon processors) interconnected by Gigabit Ethernet.

Keywords and Phrases: aspect-oriented programming, plumbing, performance specification, runtime systems, parallel programming tools

1 Introduction

The development and optimization of complex cluster applications is difficult with current technologies for parallel program development. Even when applications are well-suited for cluster computing there are several issues to be addressed in developing such applications and experimenting with them on a cluster. Consider the problem space of interactive multimedia applications. Interactive audio/video processing is rich in pipelined, task, and data parallelism. Developing and optimizing such applications involve two major issues.

The first issue is specifying how the different threads and data abstractions of the application are *connected* to one another. Commonly referred to as the *plumbing* problem in parallel and distributed computing, this connection specification is difficult to manage as the application scales up. Although threads could autonomously create the requisite data abstractions and connections, writing the routines to do this on a large scale can easily be error-prone. Adjusting these routines as the application evolves introduces further opportunity for coding errors.

The second issue is exploring different parallelism strategies for maximizing performance. Due to the potential for exploiting different pipelined, task, and data parallelism opportunities in this class of applications, the space of strategies to be explored can be quite vast depending on the scale and complexity of interactions and the number of simultaneous interactions. Furthermore, each distinct strategy leads to a distinct underlying plumbing configuration. Therefore, exploring parallelism opportunities greatly exacerbates the plumbing problem for this class of applications.

Analyzing these issues from the perspective of Aspect-Oriented Programming indicates the need for their separation from the algorithmic design. Treating the plumbing and performance characteristics as *aspects*, and the thread implementations as *components* makes it possible to separate the cross-cutting concerns. The components' implementations can remain static while the plumbing and performance aspects are varied for evolutionary and optimization purposes.

Aspect-Oriented Programming, as described in [3] is motivated by the difficulties in specification of different *cross-cutting* characteristics within the application language. They present examples of such characteristics including compile-time memory allocation constraints and loop fusion properties. Manually embedding both of these constraints within the application code could easily lead to their entanglement. The result would be a combination of the application algorithm and the particular optimization details necessary to satisfy both constraints. To ease the development and modification of such applications in the presence of "cross-cutting" characteristics, they propose the use of Aspect-Oriented Programming (AOP). In AOP, units of the algorithm's functional decomposition correspond to *components*, and other properties that cannot be cleanly expressed in an algorithmic procedure correspond to *aspects* and tend to affect the performance or semantics of components in systemic ways. Once this separation has been made, the application programmer can vary performance characteristics independently of the components. AOP includes the use of a *weaver* tool that automates the combining of component and aspect specifications to yield the resulting application that adheres to the aspects' constraints while executing the algorithm.

We have developed an *aspect language* called *STAGES* that provides aspect-oriented support for cluster computing. STAGES addresses both the plumbing and performance optimization issues identified earlier. Using this language (and its associated weaver to integrate aspects with components in an automated fashion) disentangles algorithm design from connection management and performance concerns. Since it is often convenient to represent interactive multimedia applications as coarse-grain dataflow graphs, STAGES includes a simple graph specification syntax and semantics for specifying the connections among threads and data abstractions. To address the performance optimization issue, STAGES provides a way of choosing different parallelization strategies to accomplish a particular computation, along with a way of specifying the mapping of resulting threads and data abstractions onto nodes of the cluster.

Although we have chosen a particular cluster programming support library called *Stampede* [8] in our current implementation of STAGES, the language is general. Targeting a different set of abstractions is simply a matter of specifying a different grammar for STAGES and modifying the code generation part of the weaver.

The contribution of this paper is a presentation of our aspect language, STAGES, its implementation, and its utility for mapping complex applications onto a cluster. We include performance results from exploring the parallelism space for two multimedia applications.

Section 2 provides a motivating example for separating cross-cutting concerns in a simple distributed application. In Section 3, we discuss related work. Section 4 describes the STAGES language and its current implementation targeting a particular cluster programming library called Stampede. Section 5 describes our experiments that study the utility of STAGES in the context of exploring different parallelization and plumbing choices for two different complex cluster applications. It also contains the performance results and lessons learned from these experiments. Section 6 presents concluding remarks and directions for future research.

2 Motivating Example

To motivate the use of aspect-oriented programming, we start with a simple example of the plumbing problem involving *processes* and *sockets*. Consider the code necessary to setup and manage multiple sockets between processes in a distributed application. In a functional decomposition of such an application, processes and sockets would be the nodes of a bipartite directed graph (see Figure 1). The arcs in the graph represent the coupling of processes to sockets. A process node is a "client" to a socket node if the arc originates at the process node and ends at the socket node. For example, P1 is a client of S1. Similarly, P2 is a "server" node to S1 because the arc originates at S1 and ends at P2. Note that "server" process nodes are abstractions of forking server processes.

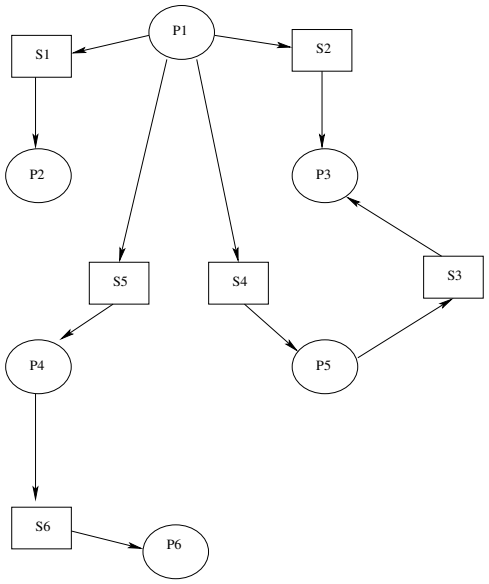

Figure 1: STAGES Example for Specifying Process/Socket Graph

Manually coding the routines to setup a single configuration of such a client/server application is difficult enough when trying to incorporate cross-cutting concerns. Some examples of such concerns are the mapping of processes to computational resources and characterizing the socket abstractions for varying performance targets, such as how many pending connections should be queued prior to acceptance.

For large scale instances of such applications, the use of aspect-oriented programming would greatly aid in development. For example, let the process implementations be the *components* and let the functional decomposition of the application be described in an aspect language that also provides the ability to determine the aforementioned mappings and socket characterizations. A compiler or preprocessor for this aspect language could generate code to bind components with aspects. This code could (1) start the processes on the requisite computational resources, (2) provide wrappers to the server processes to automate socket setup, connection acceptance, and forking the server components, and (3) provide wrappers to the client processes to automate connection to server sockets.

Using such a framework would relieve the application programmer from coding the details of connection setup for every combination of client and server socket, and would limit the scope of code changes necessary for

different graph configurations, different mappings to computational resources, and different socket characterizations.

3 Related Work

To date, there has not been a significant adoption of AOP for building large software systems [7].

Edward Lee [5] points out the importance of inter-operability of different programming frameworks and the need for well-defined compositional interfaces to combine computational models for future embedded software. He argues that without such well-defined interfaces it would be difficult to characterize the aggregate property of a software subsystem that consists of multiple components. There have been instances of domain specific languages that researchers have developed that are in the spirit of aspect-oriented programming. Devil [6] is an interface definition language that allows decoupling of the specification of hardware (such as an Ethernet controller) from the coding of its device driver. Such a decoupling allows development of less error-prone, more understandable, and re-usable device driver code. Knit [12] is a component composition language for specifying the linking requirements of separately compiled system software components. Knit helps detect subtle errors in component composition and allows the development of modular, performance conscious software.

Dataflow computation models and functional programming languages for supporting such models have been attractive for their expressive power and semantic cleanliness. However, their adoption for general-purpose computing has not been widespread due to the relatively slow implementation of these languages compared to traditional languages. There have been proposals to improve the efficiency of implementing such languages especially in a parallel setting [9]. Regardless of their adoption or lack thereof for general-purpose computing, such languages serve as inspiration for thinking about expressing cross-cutting aspects of interest to a computation in a semantically clean way. In fact, graph-based specification of STAGES is heavily influenced by this thinking.

The closest related work to STAGES is the Naval Research Laboratory's Processing Graph Method (PGM) and its associated tool (PGMT). PGM [2] targets portability of such algorithms across different multiprocessor architectures by using an architecture independent language to describe the algorithms. PGM is similar to STAGES in two ways. First, it uses a graph specification language to describe the functional decomposition of algorithms. Second, it has the ability to "split" a dataflow, process units of the split concurrently, and "join" the resulting dataflows. Unlike STAGES, PGM does not neatly decouple the performance characteristics of interest to an application (such as address space mappings for threads) from the algorithmic specification. Therefore, PGM does not provide an aspect-oriented language framework for application development.

4 STAGES Language

The STAGES language work is inspired by the desire to automate the aspects of connection management and performance specification in complex distributed and parallel applications as evidenced by the above process/socket example. However, the context for STAGES is different from this example. Instead of simple sockets and processes, we have chosen to capture the Stampede cluster [8] library abstractions in our functional decomposition and performance specifications[1]. Hence the name STAGES: STampede Application Generator Specification.

In a nutshell, the language provides primitives for instantiating computational abstractions, their connectivity, and performance strategies for mapping them onto computational nodes. The input to STAGES are: *grammar* that identifies the syntax and connection semantics of the computational abstractions; and an *instruction set* that identifies the API calls for these abstractions. The former is used by STAGES for syntax checking, and error reporting, and the latter for code generation.

Although we have targeted STAGES for a specific underlying set of abstractions (Stampede's), our language and "weaver" preprocessor has more general applicability. Modifications to the abstraction grammar, connectivity analysis, and associated code generation modules could be themselves an aspect specification. The same

[1]It should be noted that the language mechanisms needed to capture the process-socket example in Figure 1 are a subset of the capabilities currently available in STAGES.

underlying idea of separating the connection management and performance characteristics from the algorithmic components to support automated application building from disentangled concerns would apply.

Since STAGES uses Stampede's API as the target for its code generation, it is important to have some familiarity with the computational abstractions of Stampede before we discuss the language primitives of STAGES.

4.1 Stampede Programming Abstractions

Stampede programming system [8] allows the creation of any number of address spaces in a cluster[2] (at program startup) and the dynamic creation of threads that execute in any of these address spaces. It provides a data sharing abstraction, called *space time memory* (STM) [10], to enable simple and efficient management of a collection of time-sequenced data items (of arbitrary size and structure) transparently across a cluster. The STM data sharing abstraction is composed of three entities: *channels*, *queues*, and *registers*. From the point of view of an application program, every instance of these entities is a cluster-wide unique name. The API calls in Stampede for the dynamic creation of (in any of the Stampede address spaces), destruction of, connecting to, disconnecting from, and doing input/output on these entities are similar as well.

At a high level, the computational graph of a Stampede application looks similar to the process-socket graph shown in Figure 1, with the processes replaced by Stampede threads and the sockets replaced by the entities provided by STM. However, the connectivity rules for Stampede are different from the simplified processes/sockets example. There is no limit (besides system maximum values) to the number of input and/or output connections a Stampede thread can have to the STM entities. As in the processes/sockets example, in addition to the functional decomposition specification, there would need to be a specification for the mapping of graph nodes into Stampede address spaces and for the characterization of properties of the Stampede data abstractions.

While it is not essential to know the semantics of the three STM entities for appreciating the language primitives of STAGES, we present the salient differences among them for the sake of completeness. A Stampede channel holds time-sequenced data items. Typically, a channel would be used by an application to record the time-stamped output of a particular activity (modeled in Stampede by either a single thread or a collection of threads). For example, the data items may be successive images from a camera each with a distinct timestamp. For this reason there is at most one item with a particular timestamp in a given channel. A Stampede thread can make an input connection to a channel and retrieve items randomly from the channel by specifying the timestamp associated with the item (the timestamp can also be a wildcard such "latest", and "earliest"). A Stampede queue is similar to the channel in that it holds time-sequenced data items, with two main differences: a thread can retrieve items strictly in FIFO order; and there can be multiple items with the same timestamp. This entity is primarily for exploiting data parallelism in an application. For example, a single camera image may be partitioned into several chunks (all with the same timestamp) that are stored in a Stampede queue and then analyzed in parallel by several threads. A Stampede register is an entity that is like a processor register in that writing into it over-writes any previous item that was in that register. This entity would typically be used by an application as a rich event signaling mechanism among the Stampede threads.

4.2 STAGES Language Primitives

We have developed a language that incorporates the functional decomposition in a graph specification, the mapping of graph nodes, and the characterization of data abstraction properties. Our language also includes the ability to specify different *strategies* for accomplishing a particular computation node. We provide three classes of strategies: a single *simple thread*, an instantiation of a *subgraph*, and a generic *data parallelism construct*. The choice of a particular strategy and the mapping of the graph nodes to address spaces are part of the *performance* specification of the language. The reserved words associated with specifying these strategies are the only intrinsic tokens in the lexicon of the STAGES language.

Although the functional graph specification is conceptually distinct from the performance specification, we found a way to cleanly combine the two specifications to simplify our pre-processor development. Specification of the mapping of address space identifiers to physical machines is yet another aspect that properly belongs in the

[2]There is also a distributed version of this system called D-Stampede [1].

language. However, for this incarnation (i.e. for STAGES), such a specification would be completely redundant since Stampede system already gets this information from a separate configuration file at runtime.

We now present in tutorial fashion (through a series of examples) our current STAGES grammar in which the performance specification is combined within the functional specification. Note that this language is our aspect language and that the actual implementations of thread nodes (components) is in C. It should be emphasized that while we use the Stampede data abstractions (queues, channels, registers) to make these STAGES examples concrete, the language itself is agnostic about the semantics of these abstractions. STAGES simply worries about the plumbing among these abstractions and the strategies for mapping them onto address spaces (and hence onto the computational nodes) from the point of view of performance. In the examples, the reserved words in the lexicon of STAGES are shown in **boldface** font to distinguish them.

The Functional Specification must be defined within a special tag:

```
1.  functional {
2.      // Functional Specification goes here
3.  }
```

As described above, the functional specification is a graph specification language. The primary unit of specification is a *graph*, that encapsulates a unit of multi-threaded computational data flow. A graph can contain an instance of another graph (called a *subgraph*) to perform a computation. This enables hierarchical functional decomposition. To make the code easier to read and compile (in one pass), a graph can only instantiate subgraphs whose definitions lexically precede them in the specification.

4.2.1 Simple Example

```
1.  functional {
2.      graph fancy_producer(out queue q) {
3.          queue initial_images;
4.
5.          t1 computation capture_images((out) initial_images);
6.          t2 computation color_to_bw((in) initial_images, q);
7.      }
8.
9.      graph main() {
10.         typedef computation producer(out queue q) {
11.             producer_thread(q),                    // strategy 1
12.             graph fancy_producer(q);               // strategy 2
13.         }
14.
15.         queue produced_images;
16.
17.         p producer(produced_images) strategy 2;
18.         c computation consumer_thread((in) produced_images);
19.     }
20. }
```

Figure 2: Stages Example Code for Specifying the Plumbing for a Producer/Consumer Program

Figure 2 is a simple example of STAGES code that specifies a producer-consumer pipeline. Figure 3 is a visual representation of the same example (top half of the Figure is when strategy 1 is selected, and bottom half is when strategy 2 is selected). With reference to Figure 2, it is expected that the application programmer will provide C code for the thread functions "capture_images", "color_to_bw", "producer_thread", and "consumer_thread".

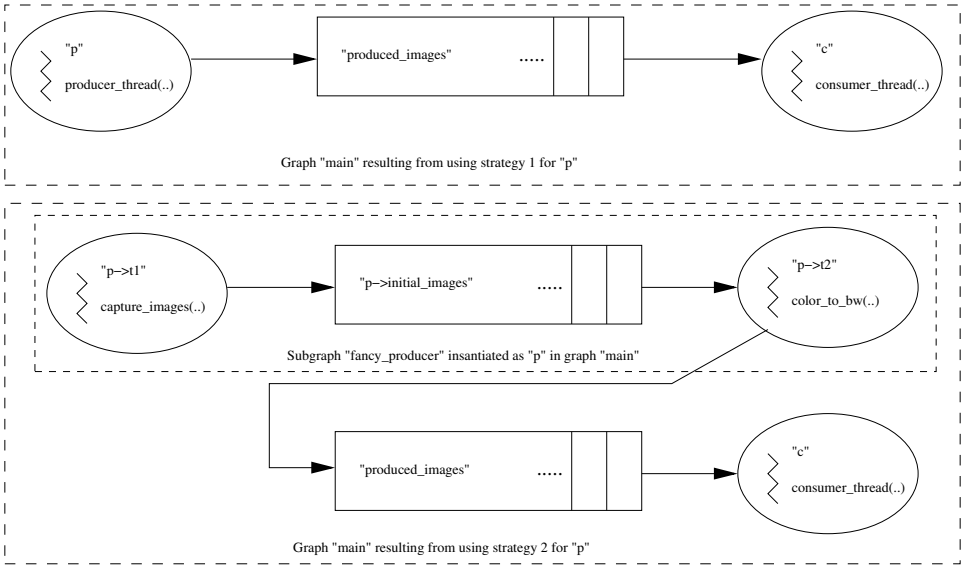

Figure 3: Producer/Consumer Program Showing the Plumbing Resulting from the Specification in Figure 2

Line 9 starts the definition of the graph called "main" (a reserved word) that is used to name the outermost encapsulating graph in the specification. Lines 10 through 13 describe 2 different strategies for being a "producer". The first strategy (line 11) is to simply use the single thread component called "producer_thread" and expect it to take a single output connection to a queue as a parameter. Note that the application programmer would need to provide the code to implement "producer_thread" in a separate C file if this strategy were used. The second strategy (line 12) is to instantiate the "fancy_producer" subgraph and expect it to perform the necessary computations to output produced data items to a queue. Note that multiple instantiations of a particular graph (except for graph "main") or thread function are allowed. Just as threads can be directly instantiated (as in line 18), so can graphs. If one wanted to always use producer strategy 2, line 17 could be replaced with "p graph fancy_producer(produced_images);", the producer typedef (lines 10-13) could be omitted.

The actual choice of which strategy to use is a performance specification. We incorporated this specification within the functional specification. The first strategy is used if unspecified. In this case, however, line 17 instantiates the second strategy, binding the formal variable "q" on line 10 to the actual variable "produced_images". As a result of this instantiation, two thread components are launched: "capture_images" and "color_to_bw". They are given the requisite connections to the "initial_images" queue declared in line 3 and the "color_to_bw" thread is also given an output connection to the "produced_images" queue. Regardless of either strategy choice, a "consumer_thread" thread is created and given an input connection to the "produced_images" queue.

The labels given to nodes in Figure 3 (*e.g.,*"p", "c", "p->initial_images", etc.) correspond to those given in the specification. If we were to ignore the possibility of a separate specification section (performance, for example) needing to refer to named graph nodes, and if we were to ignore the potential for using the labels for runtime reflection and debugging, we would not need the labels in the grammar. Besides strategy specification, the mapping of graph nodes to address spaces and attributes (*e.g.,*buffer capacity) of data abstractions can also be specified. An example of these features is given in the next subsection along with specification of data parallel strategies.

4.2.2 Specification of Data Parallel Strategies

Many applications display some form of data parallelism. For example, in a vision processing application, one may be looking for several different objects in an image frame. In this case, it is conceivable that several data parallel threads could be put to work, each looking for a specific object in a given frame. In most such situations, the best strategy in terms of number of worker threads to create, the number of work units to create,

and how to map them to different nodes of the cluster may not be readily obvious [4]. This calls for quickly and reliably generating several instances of the graph for experiments targeted to converge on an optimal strategy. The primitive for data parallel strategy specification in STAGES is inspired by this need. Typically, an application thread (a *splitter*) would create work units for the workers, and another application thread (a *joiner*) will gather the partial results from the workers to create a composite result.

```
1.  functional {
2.      graph main() {
3.          typedef computation avg_image_regions(in queue digQ,
4.                                                out queue dispQ)
5.                                                [AS x, AS y, ...] {
6.              (splitter_image, worker_avg_image, joiner_image) [x, ..., y];
7.          }
8.
9.          queue digitizerQ <5> [AS0];
10.         queue displayQ <5> [AS1];
11.
12.         dig_thr computation digitizer_bttv((out) digitizerQ) [AS0];
13.         disp_thr computation display_image((in) displayQ) [AS1];
14.         avg avg_image_regions(digitizerQ,displayQ)
15.             strategy 1 pieces 49 workers 3 outchannels 49 outqueues 0
16.             <10,4,1> [AS0,AS1,AS2,AS3,AS4];
17.     }
18. }
```

Figure 4: STAGES Example Code for Specifying a Data Parallel Strategy

Figure 4 illustrates the STAGES primitive for a generic data parallelism strategy specification via an example. See Figure 5 for a visual representation of the same example. First let us discuss how STAGES maps graph nodes to address spaces. The specifications between square braces "[...]" are address space parameterizations. For example, graph main's digitizerQ will be instantiated on address space 0, and graph main's displayQ will be instantiated on address space 1. Stampede provides a separate resources configuration mechanism to map these address space numbers to physical machines (through a configuration file). In Figure 4, address space binding during a typedef is shown. The address space variable "x" for "avg_image_regions" is bound to address space 0 when "avg" is instantiated in lines 14-16. Similarly, "y" is bound to address space 1, and "..." is bound to the sequence of address spaces 2,3,4. Note that, if present, "..." must be last in the sequence of typedef address space formal parameters. Address space parameters can be passed as actual parameters to subgraph instantiations as well. This facility is not shown in this example since there are no subgraphs ("main" implicitly starts on address space 0). However, the parameter passing mechanism during a subgraph instantiation is very similar to the binding of address space parameters during a typedef's instantiation.

The example also illustrates an optional performance specification of the desired buffering capacities for the Stampede queue and channel entities. The "<5>" on lines 9 and 10 indicates that both the digitizerQ queue and the displayQ queue should be bounded with maximum buffering capacity of 5 (omission of this specification will result in STAGES defined default buffering capacities). Similarly, the "<10,4,1>" on line 16 characterizes the capacities of the splitter output queue (10), each of the worker output channels (4), and each of the worker output queues (1) implicit in the generic data parallelism construct's instantiation.

Strategy 1 (the only strategy listed) for "avg_image_regions" on line 6 is a *generic data parallelism construct*. Such a construct is minimally specified by 4 values: pieces, workers, worker output channels, and worker output queues; and 3 components: a splitter, a worker, and a joiner. Upon instantiation, any typedef input connections are connected (by default) to an instantiation of the splitter component, any typedef output connections are connected (by default) to an instantiation of the joiner component, and the following plumbing is created:

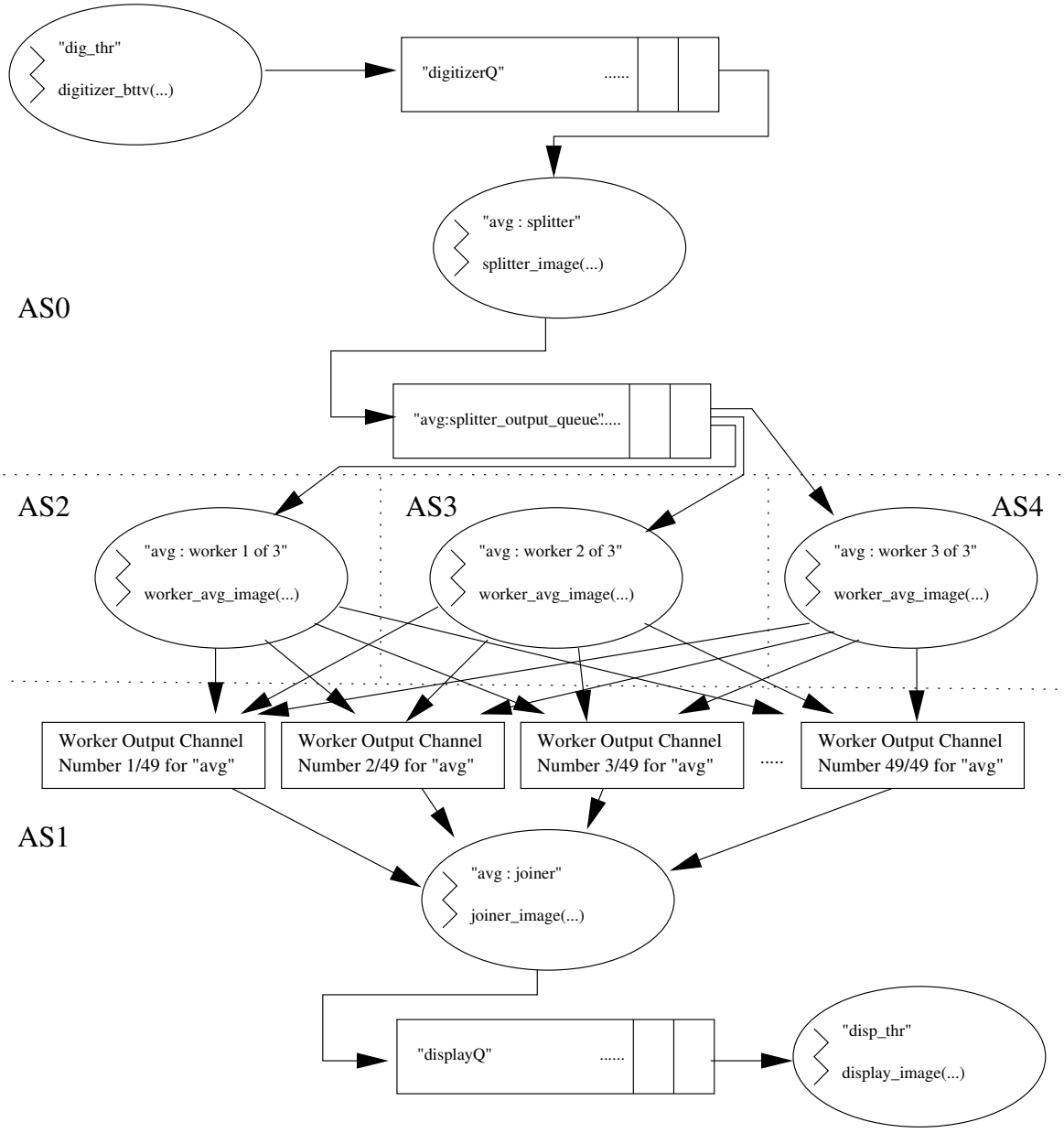

Figure 5: The Plumbing Realized by the Data Parallel Strategy Code in Figure 4

1. An instance of a queue called the "splitter output queue" is created.
2. The splitter thread is given an output connection to the splitter output queue.
3. The "workers" value is used as the number of instances of the worker component to create (3 in this example).
4. The "outchannels" value is used as the number of "worker output channels" to instantiate (49 in this example).
5. The "outqueues" value is used as the number of "worker output queues" to instantiate (none in this example).
6. Every instantiated worker is given an input connection to the splitter output queue.
7. Every instantiated worker is given an output connection to every worker output channel and queue.
8. The joiner is given an input connection to each worker output channel and queue.
9. Each splitter, worker and joiner is given the "pieces" value as a parameter.

In addition to this minimal specification of a *generic data parallelism construct*, STAGES allows address space parameterization in the instantiation of this construct. The mapping of the implicitly generated graph nodes to address spaces using the specified parameters is summarized in Table 1.

AS Param Format	Splitter	SplitOutQ	Workers	WorkerOutputChannels/Queues	Joiner
None	AS0	AS0	AS0	AS0	AS0
$[v]$	$[v]$	$[v]$	$[v]$	$[v]$	$[v]$
$[v, w]$	$[v]$	$[w]$	$[w]$	$[v]$	$[v]$
$[v, ..., w]$	$[v]$	$[v]$	round-robin $[...]$	$[w]$	$[w]$

Table 1: Performance Specification in STAGES. Note that v and w can be either AS param variables from the typedef AS parameter formals list or literal address space values such as AS10.

4.2.3 Extensions to Generic Data Parallelism Construct

In addition to the implicitly created graph nodes due to the data parallelism construct, STAGES allows the specification of explicit connections to the splitter, all workers, and the joiner. For example, suppose the avg_image_regions strategy 1 shown in Figure 4 is to be augmented to allow (a) each of the workers to have a direct input connection to an external channel that gets data from the digitizer, and (b) the splitter to be able to communicate via a private channel to the joiner. The specification could be modified fairly simply to accommodate these constraints as shown in Figure 6.

Note that digQ needs to be explicitly listed in line 9 now that we are listing other parameters to the splitter. The same applies to the joiner in line 11. Further performance optimization might involve determining the best address space placements and capacities for digitizerC and split_to_joiner.

The two applications we examine in Section 5 exhibit different kinds of data parallelism for which we use this generic data parallelism construct as part of the specification.

```
1.  functional {
2.      graph main() {
3.          typedef computation avg_image_regions(in queue digQ,
4.                                                out queue dispQ,
5.                                                in channel digC,
6.                                                out channel split_to_joiner_o,
7.                                                in channel split_to_joiner_i)
8.                                                [AS x, AS y, ...] {
9.              (splitter_image(digQ,split_to_joiner_o),
10.              worker_avg_image(digC),
11.              joiner_image(dispQ,split_to_joiner_i)) [x, ..., y];
12.         }
13.
14.         queue digitizerQ <5> [AS0];
15.         queue displayQ <5> [AS1];
16.
17.         channel digitizerC;
18.         channel split_to_joiner;
19.
20.         dig_thr computation digitizer_bttv((out) digitizerQ,
21.                                            (out) digitizerC) [AS0];
22.
23.         disp_thr computation display_image((in) displayQ) [AS1];
24.
25.         avg avg_image_regions(digitizerQ,displayQ,
26.                               digitizerC,split_to_joiner,split_to_joiner)
27.             strategy 1 pieces 49 workers 3 outchannels 49 outqueues 0
28.             <10,4,1> [AS0,AS1,AS2,AS3,AS4];
29.     }
30. }
```

Figure 6: Modified Data Parallel Strategy Code with a Private Channel with Joiner

4.2.4 Discussion

The series of examples in this section illustrates the expressive power of STAGES for capturing the plumbing and performance specification aspects of parallel applications. Further, even these simple examples are sufficient to show that the search space for performance optimization is vast. For instance, the modification of the application from Figure 4 to Figure 6 may imply that we need to change the numbers of pieces and workers, address space mappings or even capacities of the various STM entities to achieve optimal performance.

When such modifications are made to the functional composition and performance specifications of an application, errors in connection specification could be introduced, especially if there were no language and tool to assist in the evolutionary process as STAGES does. To help minimize errors at the level of functional specification, our STAGES pre-processor performs connectivity analysis to ensure that every instantiated abstraction is used (*i.e.,*has at least 1 input connection and at least 1 output connection). We have introduced special keywords into the grammar to explicitly allow a data abstraction to be either a "sink" (no threads have input connections to it) or a "source" (no threads have output connections to it) in case the application programmer is determined to circumvent the connectivity analysis. To inform the application programmer about potential errors in the specification, our pre-processor can be asked to give warnings for graphs that are never instantiated and for performance characteristics (capacities and address space mappings) that are unspecified. To improve readability of STAGES, we include the traditional C++ commenting styles.

4.2.5 Implementation

Our implementation of STAGES is currently in the form of a pre-processor composed of flex, bison and C code. The result, called *stage*, generates C code from STAGES input. The generated code effects the runtime instantiation of the specified STM entities, computations and connections. Furthermore, these instantiations are done with respect to the specified performance characteristics. The application programmer need only create the C code that performs the computations, create the STAGES describing how the application is composed including the desired performance characteristics, and compile and link these with the cluster Stampede runtime libraries to produce a runnable Stampede cluster application.

5 Exploring the Power of STAGES for Performance Analysis

The focus of this section is to show the power of STAGES for exploring the parallelism space of compute intensive applications. We have chosen two applications: *video textures*, and *color-based people tracker*. Both use video as input. Video textures [13] takes as input a finite sequence of video frames of a repetitive scene (*e.g.,* a flag waving in the wind, a child swinging in a swing-set, etc.) and generates an infinite sequence of video. The core algorithm in this application is a difference calculation between every pair of images in the input sequence. The space of parallelism to be explored for this algorithm is the distribution of subsets of images to a particular node of the cluster for analysis, and the number of segments into which a single frame is broken up.

In the color-based people tracker [11], continuous video input is analyzed to locate people in the scene based on some specific attribute (*e.g.,* shirt color). The parallelism space to explore in this case is vast and includes the distribution of the pipeline of threads that make up the computation (such as digitizer, background subtraction, histogram generator, and tracker) to cluster nodes, the number of color models to look for in a frame, and the number of segments into which a single frame is broken up.

Since the focus of this section is not the performance of the applications *per se*, but the expressive power of STAGES to explore the parallelism space, we restrict our exploration to two kinds of data parallelism. For the video textures application, the strategy we explored is the distribution of subsets of images to different cluster nodes. For the tracker, the strategy we explored is the segmentation of each image into several fragments. The application pipelines are shown pictorially in Figures 7, and 8, respectively. STAGES codes for these two applications are similar in succinctness to the examples shown in Section 4, and parameterize the parallelism strategies to be explored.

We present the results of experimenting with the two applications in the next two subsections. The experimental platform is a 17-node cluster, wherein each node is an 8-way SMP using a 550MHz Pentium III Xeon processor with 4GB RAM and 18GB SCSI disk. The cluster interconnect is Gigabit Ethernet. Each cluster node runs RedHat Linux 7.1. As we mentioned earlier, the STAGES code gets compiled into C with calls to the Stampede library. For inter-node communication, Stampede uses a reliable messaging layer called CLF which sits on top of UDP.

5.1 Results for Video Textures

For the video textures application, we use a set of 315 input images (each image is 640 x 480 pixels). The analysis of performance for the video textures application is done in two ways to examine application behavior with and without communication of data from the producer to the consumers.

5.1.1 Producer Preloaded

In this experiment, the producer thread loads all of the images and then commences timing and distribution of images to the consumers for processing. As in all of these experiments, each address space corresponds to a distinct 8-way SMP cluster node. The producer runs in an address space distinct from the consumer address spaces to force the distribution of images to involve communication between address spaces. The timing completes when all consumers indicate to the producer that they are done processing.

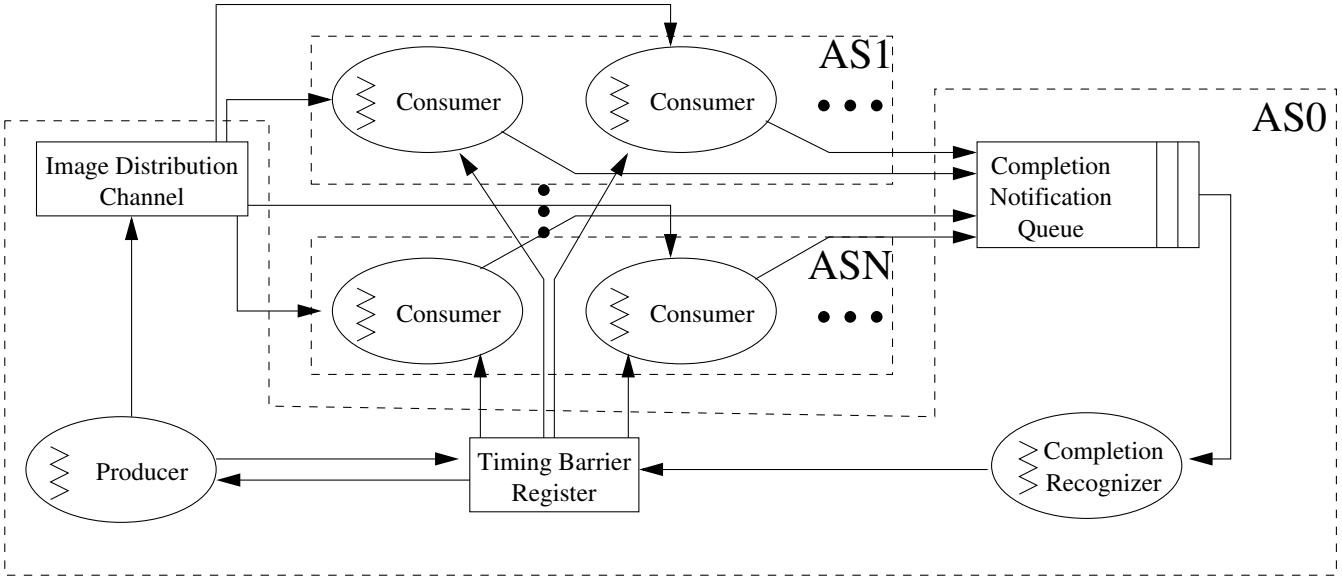

Figure 7: Video textures application pipeline with two dimensions of scale *i.e.,* number of consumer address spaces and number of threads per consumer address space

Number of Consumer Address Spaces	Consumer Threads per Address Space									
	1	2	3	4	5	6	7	8	9	10
1	123	86	66	54	55	49	44	44	44	51
2	68	55	46	40	38	35	33	33	31	31
3	63	49	41	35	32	30	29	28	28	28
4	51	38	31	30	30	29	28	28	30	31
5	40	32	31	29	29	29	28	28	28	27
6	43	37	34	32	31	31	30	30	30	30
7	49	41	38	36	35	35	35	35	35	35
8	44	37	35	36	35	35	36	35	36	35

Table 2: Video Textures, Producer Preloaded, Time to Complete Processing (seconds)

Table 2 summarizes the results for 80 different configurations. Each configuration requires a completely different plumbing among the computational entities shown in Figure 7. The numbers in the table encode subtle details about the underlying cluster architecture and its interaction with application properties. Our purpose is not to analyze these results in any great detail. The intent is merely to show the capability of STAGES to enable the rapid exploration of such a vast parallelization space. As an aside, two interesting patterns can be observed by inspecting the table. First, it is cheaper to perform processing on two address spaces with one consumer thread in each than to perform processing on a single address space with two consumer threads. This pattern is continued in general along the first row and column of the result table. The second observation is that the best performance is approximately 28 seconds and the corresponding configurations do not appear to follow any obvious pattern without running this experiment. These two observations are fairly non-intuitive and could not have been surmised without actually running such experiments.

5.1.2 Consumer Preloaded

To analyze performance in the absence of communication of images, the producer commences timing in this experiment only after all consumers have independently loaded the images. In essence, this experiment turns the original application into an embarrassingly parallel application. While this is not a realistic experiment from the point of view of this application, one may want to do this just to see the limits of inherent parallelism in the

Number of Consumer Address Spaces	Consumer Threads per Address Space									
	1	2	3	4	5	6	7	8	9	10
1	98	69	52	41	43	35	35	33	31	33
2	49	39	32	27	23	20	18	18	17	16
3	47	37	26	24	20	16	15	14	14	14
4	32	24	17	13	13	11	10	10	10	10
5	22	17	12	11	9	10	8	7	8	7
6	23	18	13	11	9	8	7	7	7	7
7	23	18	12	10	8	8	7	6	7	7
8	17	13	9	7	7	6	6	5	5	5

Table 3: Video Textures, Consumer Preloaded, Time to Complete Processing (seconds)

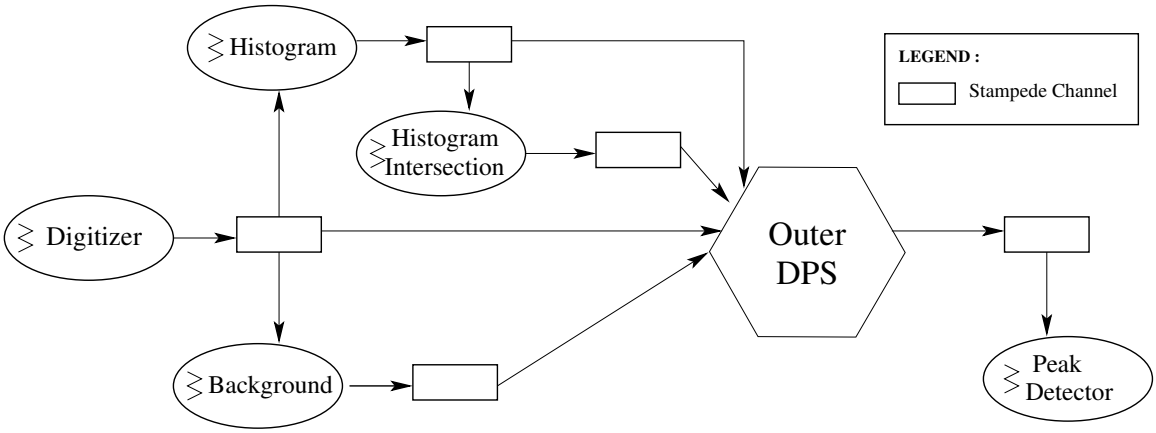

Figure 8: Tracker Pipeline: See Figure 9 for Outer DPS

application in the absence of any communication. The results are summarized in Table 3. As might be expected, for a fixed number of address spaces, adding more consumer threads increases the application performance asymptotically. For example, consider the row for 4 consumer address spaces. As the number of threads per consumer address space increases, the execution time decreases from 32 to 10 seconds. Towards the end of the row, the completion time stabilizes at 10 seconds. This represents an asymptotic upper bound for performance. Similarly, for a fixed number of consumer threads per address space, increasing the number of address spaces exhibits similar asymptotic behavior (see for example the column for 7 threads per consumer address space). However, the expected asymptotic value for a particular number of consumer address spaces or for a particular number of consumer threads is not obvious without running this experiment. Ideally, the processing should scale linearly as the number of address spaces or as the number of threads per address space increases. However, the results in Table 3 show that this is not the case.

5.2 Results for Tracker

The data parallel implementation of the tracker is quite complex as seen from Figures 8 and 9. The tracker pipeline is illustrated in Figure 8. Components in the pipeline include the following data generators (accumulators): the frame digitizer, the background thread, the histogram thread and the histogram-intersection thread. The Outer *Data Parallel Subgraph (DPS)* takes input from these threads and performs the actual color tracking by applying various parallelizing techniques described below.

The generic *DPS* is again illustrated in Figure 9. It consists of a *splitter* that breaks an image into pieces that are picked up by *workers* that can themselves be either embedded *DPSs* or simple threads that perform the actual color tracking. The *joiner* in each *DPS* then combines the image pieces with information from a *control channel* to compose the final result data.

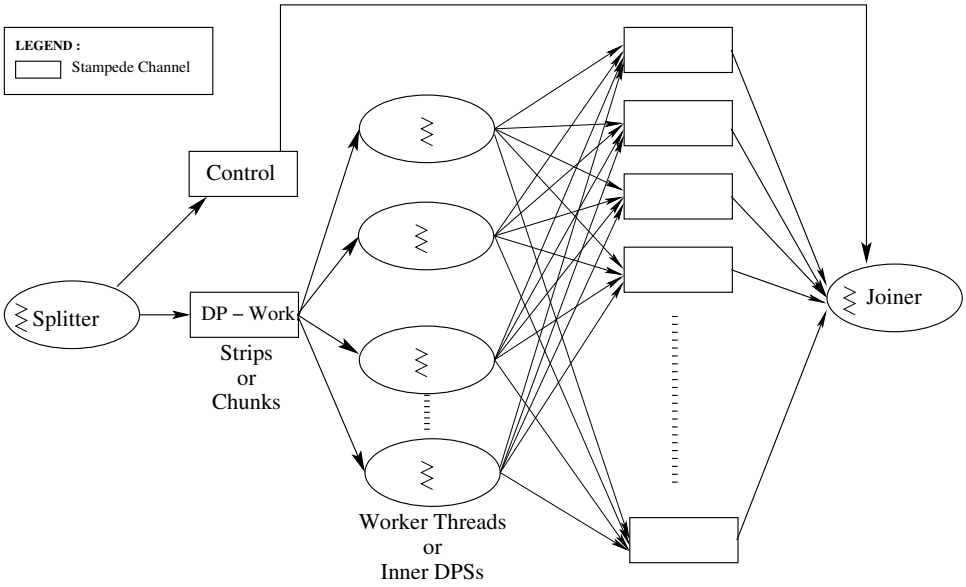

Figure 9: Tracker Data Parallel Subgraph: Used for both the Outer DPS and the Inner DPS

The Outer DPS as shown in Figure 8 is a singleton entity. The Outer DPS splitter breaks an image into *n strips* that are then retrieved by the *n* Inner DPSs. Each Inner DPS then breaks each *strip* into *m regions* which are then processed by *p* worker threads. Overall, *n*p* worker threads are instantiated that collectively work on a total of *n*m chunks*. Clearly, such a recursive data parallel pipeline creates a large space of configurations. We explore part of this space with the help of STAGES and report some results on throughput and latency performance.

Data Parallel Configuration					Throughput *(frames/second)*			
Inner DPSs (=No. of strips) *(n)*	Total worker threads *(n*p)*, *p*=4	Regions created by Inner DP *(m)*	No. of chunks *(n*m)*	Workers per chunk *(n*p)/(n*m)*	Number of Address Spaces 1	2	4	8
2	8	1	2	4	5.57	4.17	N/A	N/A
2	8	4	8	1	3.33	3.23	1.56	1.25
4	16	1	4	4	3.12	3.12	2.63	N/A
4	16	4	16	1	2.27	2.86	2.63	N/A
8	32	4	32	1	1.28	1.54	1.49	1.23

Table 4: Tracker Throughput

Tables 4 and 5 depict effects of various DP strategies on throughput and latency. Each strategy configuration is run on 1, 2, 4 and 8 Address Spaces with each AS mapped to a distinct physical node, thereby exploiting greater system resources at the expense of data-copy and network transmission overhead. Overall the results show that the application does not scale well for the parallelization space that we chose to experiment with. As in the video textures results, our purpose is not to analyze these performance results in detail. More importantly, this application shows the challenge in mapping such a complex application onto a cluster. STAGES helps in dealing with this complexity.

5.3 Summary

For both applications, manually updating the application plumbing and choice of parallelism strategy for different configurations involves effort and potential introduction of errors, whereas the corresponding changes to the STAGES specification involve simple and few updates to the connection and performance aspect configurations. STAGES allows rapid exploration of the parallelization space for an application. As a consequence, it is possible to quickly identify potential limitations to scalability and prune inefficient mappings.

Data Parallel Configuration					Latency *(milliseconds)*			
Inner DPSs (=No. of strips) *(n)*	Total worker threads *(n*p)*, *p*=4	Regions created by Inner DP *(m)*	No. of chunks *(n*m)*	Workers per chunk *(n*p)/(n*m)*	Number of Address Spaces 1	2	4	8
2	8	1	2	4	1743.88	1615.56	N/A	N/A
2	8	4	8	1	1881.16	2016.40	5998.93	7748.40
4	16	1	4	4	2015.18	2506.60	3353.76	N/A
4	16	4	16	1	2729.26	2460.58	3330.07	N/A
8	32	4	32	1	4920.71	5593.88	6396.40	7853.15

Table 5: Tracker Latency

6 Conclusion

Specification of the mapping of computational abstractions to compute nodes, and their plumbing are two important issues in the design of complex parallel and distributed applications. In traditional distributed and parallel programming frameworks, the implementations of these specifications is often interleaved with the algorithmic portions of the application. Such a design leads to poor maintainability, and lack of flexibility in code evolution. In complex applications, significant exploration of the parallelism space may be required before determining an effective mapping and its associated plumbing. To separate such cross cutting concerns from the algorithm, we have adopted the aspect-oriented programming framework.

We have designed an aspect-oriented programming language called STAGES to allow specification of plumbing and parallelization strategy selection aspects independently of the parallel/distributed algorithm code. We have demonstrated the expressive power and flexibility of the language by applying STAGES to two applications. This demonstration has shown the ease with which a vast space of task and data parallelization strategies can be explored.

Our experience in using STAGES for these applications has revealed interesting new avenues for evolving the language. For example, we plan to include the ability to express arrays of primitive data-types such as channels, queues and threads. The current STAGES implementation applies to cluster graphs. We are extending it to D-Stampede, a distributed version of the Stampede library. We are planning to expand STAGES to address other cross-cutting concerns such as QoS and failure adaptation.

References

[1] Sameer Adhikari, Arnab Paul, and Umakishore Ramachandran. D-Stampede: Distributed programming system for ubiquitous computing. In *Proc. 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002. To Appear.

[2] David J. Kaplan and Richard S. Stevens. Processing graph method 2.0 semantics, September 1995. URL: http://www.ait.nrl.navy.mil/pgmt/pgm2.html.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag LNCS 1241.

[4] Kathleen Knobe, James M. Rehg, Arun Chauhan, Rishiyur S. Nikhil, and Umakishore Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proc. SC99: High Performance Networking and Computing Conf*, Portland, OR, November 1999. Technical paper.

[5] Edward A. Lee. What's ahead for embedded software? *IEEE Computer*, September 2000.

[6] Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An idl for hardware programming. In *Proc. Operating Systems Design and Implementation*, 2000.

[7] Sandra Kay Miller. Aspect-oriented programming takes aim at software complexity. *IEEE Computer*, pages 18–21, April 2001.

[8] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *The 11th International Workshop on Languages and Compilers for Parallel Computing*, 1998. To Appear.

[9] Rishiyur Sivaswami Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. *Intl. J. of High Speed Computing*, 5(2):171–223, 1993. Presented at Workshop on Massive Parallelism, Amalfi, Italy, October 1989.

[10] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proc. ACM Principles and Practices of Parallel Programming*, May 1999.

[11] James M. Rehg, Maria Loughlin, and Keith Waters. Vision for a Smart Kiosk. In *Computer Vision and Pattern Recognition*, pages 690–696, San Juan, Puerto Rico, June 17–19 1997.

[12] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, , and Eric Eide. Knit: Component composition for systems software. In *Proc. Operating Systems Design and Implementation*, 2000.

[13] Arno Schoedl, Richard Szeliski, David H. Salesin, and Irfan Essa. Video textures. In *Proc. SIGGRAPH 2000*, pages 489–498, July 2000.